

Industrial Automation Programming Environment with a New Translation Algorithm among IEC 61131-3 Languages Based on the TC6-XML Scheme

J. Asensio^{*1}, F. Ortuño², M. Damas³, H. Pomares⁴

Information and Communication Technologies Research Centre, University of Granada (CITIC-UGR)
C/ Periodista Daniel Saucedo, Granada 18071, Spain

^{*}jasensio@atc.ugr.es; ²fortuno@atc.ugr.es; ³damas@atc.ugr.es; ⁴hpomares@atc.ugr.es

Abstract

In this work, a complete environment for the development of industrial automation programs based on the IEC 61131-3 standard and the PLCopen TC6 scheme is presented. This environment includes a specific editor for every programming language of the standard, a set of translators for these languages and a compiler to generate executable code. This paper describes the structure and features of this application, as well as its advantages over other solutions. A new method of translation into IL language is also suggested, which takes advantage of the structured information provided by the TC6-XML scheme. The transformation algorithm is designed based on Activity-On-Vertex (AOV) networks and a topological sort technique.

Keywords

AOV Network; IEC 61131-; Programmable Logic Controller; Topological Sorting

Introduction

Nowadays, the majority of industrial sectors employ Programmable Logic Controller (PLC), the electronic device and first introduced in the 60s, to automate and control their systems. It was the decrease in the prices of processor-based systems and the need to simplify the relay-based ones what encouraged the spread of PLCs.

The evolution of computer equipment and communication technologies, and their subsequent integration into the industry, have encouraged that over the last few decades, more and more computer and electrical engineers have specialized in the automation sector. This specialization, however, was not a trivial process, since these professionals had to become familiar with very specific and unknown

programming environments (Molina, Barbancho, Leon, Molina, & Gomez, 2007). As a result, a standardization was urgently required. Organizations such as IEC (International Electrotechnical Commission) or ISO (International Organization for Standardization) started considering it in the early 90s. Thus, the standard IEC 61131-3 born. In this context proposes a common interface for PLCs with five programming languages, three of which have a graphic interface: Sequential Function Chart (SFC), Function Block Diagram (FBD) and Ladder Diagram (LD), while the rest is based on plain text: Instruction List (IL) and Structured Text (ST) (John & Tiegelkamp, 2010; Öhman, Johansson, & Årzén, 1998).

In addition, the PLCopen association (Association, n.d.) aims to promote and update the standard IEC 61131. Specifically, the Technical Committee 6 (TC6), within PLCopen, suggests using XML (eXtensible Markup Language) as an interface to exchange projects between different development environments.

This paper presents an integrated development environment, which will be referred to as HEPICA, based on the IEC 61131-3 standard and the TC6-XML scheme. In HEPICA, automation programmes can be designed using any of the programming language with both graphic and textual interfaces. To generate the executable file, a conversion to a single textual version of the program is required. Therefore, one of the main challenges in this work has been the design of an algorithm that converts all graphic programmes into textual ones.

The proposed translation algorithm has a graphic diagram (FBD or LD) as input. Using AOV graphs, the

proposed methodology obtains an ordered list of the elements that make up the diagram by the utilization of a topological sort procedure (Cormen, Leiserson, Rivest, & Stein, 2001) resulting in an equivalent textual code obtained in IL.

The rest of the paper is structured as follows: Section 2 describes the IEC 61131-3 standard and its programming languages. Section 3 presents different methodologies used in this work. Section 4 proposes a new algorithm to translate graphic languages into IL. Then, section 5 gives a description on the implemented integrated development environment along with its main parts and functions. Final section 6 summarises his paper.

IEC 61131-3 Standard

The third section of the IEC 61131 standard for PLC programming languages focuses on the structure and execution of PLC programmes, as well as on the syntax and semantics used in their development.

Concerning the structure, the standard establishes that every single program is to be organized in POU (Program Organization Units). These POU are the basic programming units, and there exist three different types of them: programs, functions and function blocks. Programs constitute the highest-level of the hierarchy, and are normally assigned to tasks in charge of their execution. Function and function block POU are auxiliary procedures that could be summoned by programs to perform a specific operation.

The standard defines the five programming languages already mentioned in the previous section along with a textual representation to define common elements to all languages, such as project configuration, resources, associated tasks or variables. In the following sub-sections, the main characteristics of the programming languages involved in the translation process are briefly described.

Function Block Diagram (FBD)

FBD is a graphic programming language used to implement programmes and complex processes using function libraries. These programmes can be defined either by the standard itself or self-created and customized by the user. A function is graphically represented as a block with a specific number of input and output connections. Thus, all the functions can be

wired as a digital circuit. An example of a Function Block Diagram is shown in fig. 1.

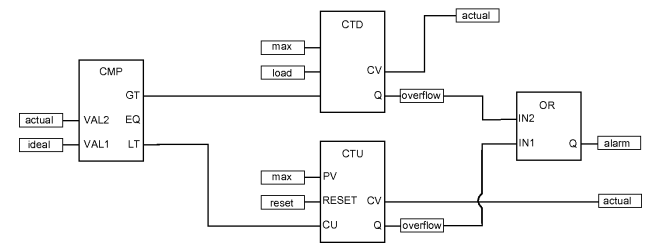


FIG. 1 AN EXAMPLE OF FBD DIAGRAM DESIGNED IN THE PROPOSED ENVIRONMENT.

Together with these function blocks, the standard defines a series of graphic elements that facilitate the design of the so-called circuit: connections, jumps, labels, variables and returns.

Ladder Diagram (LD)

A Ladder Diagram consists of a collection of graphic symbols laid out in a network, representing a logic diagram, whose input and output parameters are represented by contacts and coils respectively. It is also possible to insert function blocks among them. All these elements and the connections between them are enclosed between a right and a left power rail (See fig. 2).

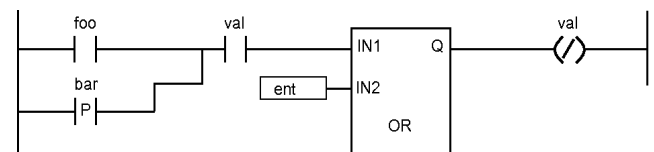


FIG. 2 AN EXAMPLE OF LADDER DIAGRAM DESIGNED IN THE PROPOSED ENVIRONMENT

Instruction List (IL)

IL is a textual low-level programming language similar to assembly language. As it can be seen in FIG. 3 that a series of text lines is inclusive of it, each of which represents an instruction. Each of these instructions is composed of a set of operands and an operation code, and it is always related to the current programme result. The operation code indicates which operation is to be performed next between this current result and the specified operands. Only one instruction per line is allowed on IL. Although IL code is more compact and executes faster than any other graphic language, most industrial control engineers do not make frequent use of this language because it is less visual than others, more difficult to debug and insufficient to program more complex functions such as PID controllers.

```
(*Multiply and divide two integers*)
ld integer1
div integer2
st result
ld integer1
mul integer2
st result2
```

FIG. 3 AN EXAMPLE OF TEXTUAL PROGRAM IN IL LANGUAGE

Methodology

Today, PLC manufacturers develop programming and configuration environments according to their own interests which happen to be very different. The proposed IEC 61131-3 standardization has contributed to making every already existing environment more compatible with the rest, but there is still a lack of full interoperability. It is on this point where the recommendation of TC6 about using XML can be applied. Thanks to XML, it has been possible to define a simple framework to translate graphic diagrams into IL language.

This conversion between the IEC 61131-3 languages was not meant firstly in the standard. New translation algorithms have just been developed in the last few years so as to enhance the use of the standard. Among all the possibilities to convert graphics into text, the LD to IL solution has been so far the most conversion (J.T. Welch, 1997; John T. Welch, 1995). One significant example is the work by (Fen & Ning, 2006, who presented an algorithm based on AOV graphs and binary trees. In a similar way, Feihu Hu et al. (Hu, Fu, Liu, & Zhang, 2008) designed a technique that transforms an AOV graph into a binary tree using a series-parallel merging methods. In (Thapa, Park, Park, & Wang, 2009), timed message based part state graphs (t-MPSG) are translated into IEC standard PLC code. Petri Nets and X-machines have also been applied as translation methods (Chambers, Holcombe, & Barnard, 2001; Lee, Zandong, & Lee, 2004). Most of these methods usually need a data structure, where information concerning different diagram elements and the dependencies between them can be defined and saved (Yan & Zhang, 2010). Nevertheless, this data structure is not required in the method proposed in this paper, since all the relevant information can be obtained automatically from the XML-TC6 scheme in which all diagrams are represented. In addition to this, the vast majority of these methods only contemplate LD diagrams made up by simple elements: coils,

contacts and power rails. Very few consider complex elements such as reset, set, complex coils and contacts or functions blocks (Yan & Zhang, 2010). LD is an easy-to-use language due to its similarity with logic circuits, but when the design of a complex LD diagram is expected, the use of complex LD elements is required as well (Plc & Engineering, 1992). Consequently, it would be advisable to include in HEPICA's LD diagram editor not only the simple elements, but also the complex ones.

On the other hand, FBD is less complicated and more easy-to-read than LD in complex diagrams. Since very little literature about FBD to IL conversion is available, it has been considered to include in HEPICA's translation algorithm the functionality required to convert FBD diagrams into IL.

Thus, the new approach proposed in this paper tries to overcome the deficiencies present in other solutions by means of the implementation of a common algorithm to translate complete LD and FBD diagrams into an IL version. In the upcoming subsections, procedures and theoretical aspects used in the design of the proposed algorithm are described in details.

TC6-XML Scheme

TC6-XML provides a basic interoperability scheme between projects designed by different environments, keeping the software independent on the underlying hardware. Furthermore, TC6 enhances the use of XML technologies such as DOM (Document Object Model), SAX (Simple API for XML) or XSL (eXtensible Stylesheet Language). Some industrial programming environments (i.e., Beremiz (Tisserant, Bessard, & De Sousa, 2007) and Multiprog (KW Software, 2012)) have already used this scheme in the persistence of their projects. Not only does the XML scheme store data concerning the project's structure or its graphic representation, but also extra information such as elements' ids or dependencies (Esteez, Marcos, & Irisarri, 2009) are stored. Since all this additional information has already been provided in the XML file, there is no need of an auxiliary data structure to keep a record of it. fig. 4 shows a TC6-XML fragment corresponding to a FBD block. In the figure, localId indicates the id of the block, and every refLocalId in connectionPointIn represents the id of its predecessors. This information is extremely helpful in the translation process.

```

<block localId="10" width="100" height="120" typename="POW">
  <position x="220" y="290" />
  <inputVariables>
    <variable formalParameter="IN">
      <connectionPointIn>
        <relPosition x="0" y="100" />
        <connection refLocalId="11" />
      </connectionPointIn>
    </variable>
    <variable formalParameter="EXP">
      <connectionPointIn>
        <relPosition x="0" y="60" />
        <connection refLocalId="9" />
      </connectionPointIn>
    </variable>
  </inputVariables>
  <outputVariables>
    <variable formalParameter="Q">
      <connectionPointOut>
        <relPosition x="100" y="100" />
      </connectionPointOut>
    </variable>
  </outputVariables>
</block>

```

FIG. 4 XML REPRESENTATION OF A FUNCTION BLOCK IN FBD LANGUAGE FOLLOWING THE TC6 SCHEME.

Activity on Vertex (AOV) Networks

An AOV network is a graph $G=\langle V, E \rangle$ formed by vertices and edges, where vertices represent tasks and edges define relations between these tasks. Vertex i is said to be a predecessor of vertex j if and only if there exists in a path from i to j in the graph. i is called immediate predecessor of j if and only if $[i, j]$ exists in E . In that case, j is called successor or immediate successor of i . Consequently, in an AOV network, for every single edge $\langle i, j \rangle$ in E , the execution of j must wait until i has been successfully executed. The AOV graph associated with the FBD diagram in fig. 1 is shown in FIG. 5

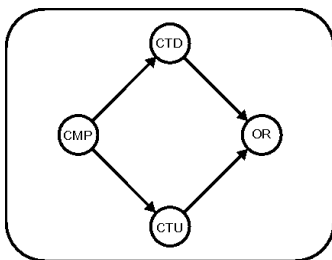


FIG. 5 A SIMPLE AOV NETWORK CORRESPONDING TO THE FBD DIAGRAM IN FIG. 1

Topological Sorting

Topological sorting is used to find out the order in which all the tasks should be executed according to the precedence restrictions imposed by the FBD and LD diagrams. Given a graph $G=\langle V, E \rangle$, this technique defines a lineal order of all the vertices in V in such a way that for every edge $\langle i, j \rangle$ in E , i will always precede j . It is therefore guaranteed that a vertex is not executed before its predecessors.

As a result, an ordered vector in which none of its elements is executed before its predecessors is obtained. fig. 6 shows the resulting vector after ordering the AOV network in FIG. 5.

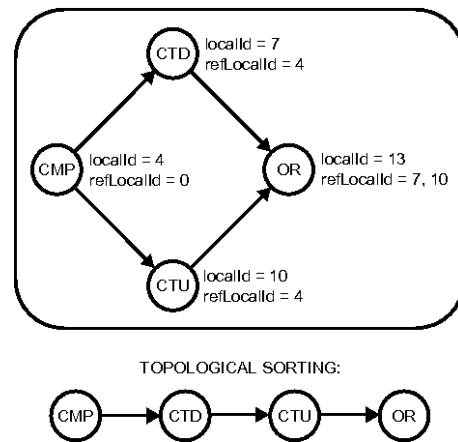


FIG. 6 RESULTS FROM THE TOPOLOGICAL SORT PROCEDURE APPLIED TO THE FBD DIAGRAM EXAMPLE.

Translation Algorithm

Prior to the execution of the translation algorithm, a transformation of the program into an AOV diagram must be carried out. In order to design the AOV net, block elements (in FBD) and blocks, coils, and contacts (in LD) will be considered as possible nodes. The relation of predecessor-successor between nodes will constitute the edges of the AOV net. The rest of elements (variables, jumps, labels, connections, etc.) are ruled out and therefore, they are not used to create this graph.

Once the AOV diagram is built up, the translation process takes place which is divided into three well-differentiated phases: Division into independent components, topological sorting and translation to IL. Those elements previously ruled out will be again considered in the final translation phase.

Division into Independent Components

In an AOV graph, there are usually several groups of nodes without connections or dependencies with the rest. Every independent group called a component will be analysed as a unique graph.

The aim of this phase is to differentiate among the independent sections that normally coexist on graphic diagrams. These sections, however, may not be easily differentiated, so a sorting process need be applied to the graph so as to facilitate this identification.

The phase starts by defining some data structures. A vector called X will contain all the nodes existing in

the AOV. A matrix M will store the independent components. Each row of M will represent a different component, whereas each column will represent nodes. This matrix could contain *null* values if necessary.

The translation algorithm starts once these structures are created. It proceeds as shown in the data flow diagram of FIG. 7. The diagram shows how the nodes of vector X are taken one by one and how, according to their dependencies on other nodes, they are placed into matrix M .

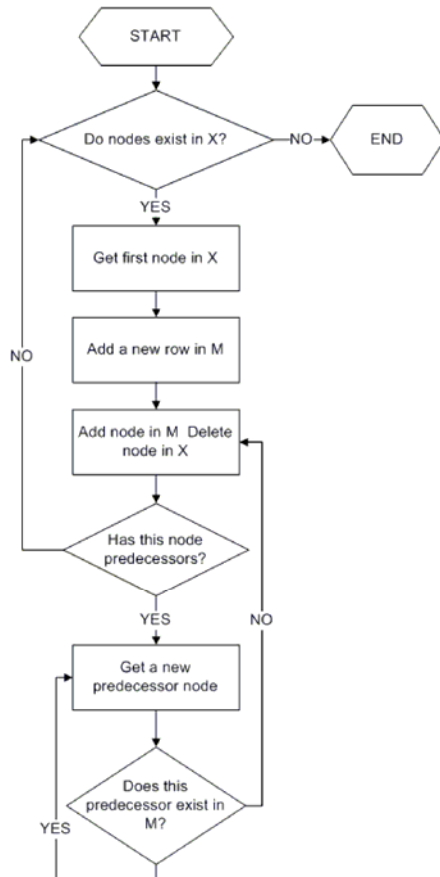


FIG. 7 ALGORITHM TO DIVIDE THE AOV NET INTO INDEPENDENT COMPONENTS.

Topological Sort Process

The ordering of all the components in the AOV graph is the most critical and important stage during the translation process. Each independent component is sorted separately. The order in which these components will be executed is from left to right and top to down. However, the ordering of the nodes in each component must follow the guidelines of the topological sorting.

In the sorting of a specific component, a new vector D is defined. This array will store the remaining unsorted nodes of this component. The vector O will contain the same nodes after being ordered. The

algorithm that turns the unsorted vector V into a sorted one is shown in FIG. 8. The methodology searches those nodes that do not have predecessors or those nodes whose all predecessors have already been evaluated. After a node and all its predecessors has been contemplated, it changes immediately from D to O . The process ends when D gets empty.

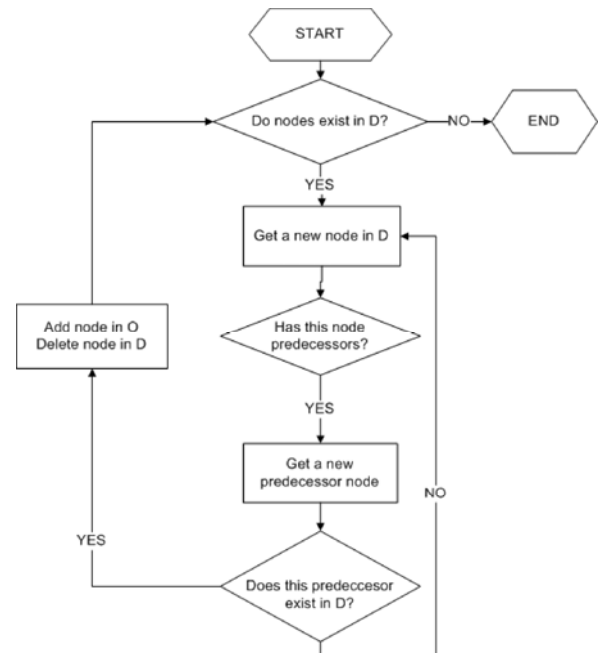


FIG. 8 FLOW DIAGRAM REPRESENTING THE ALGORITHM FOR THE TOPOLOGICAL SORTING OF THE AOV NODES.

Once the components are sorted, the algorithm, in its next phase, will run through every node in the net and will execute the function associated with each one of them. To execute one function, it is essential to know in advance the result obtained after its predecessor functions or nodes have been executed. Therefore it is of extreme importance that this sorting process is perfectly done. Otherwise, an erroneous translation into IL could be obtained.

Translation into IL

At this point, all independent components, as well as every node in them, are sorted. Now all these nodes must be translated into their respective IL function. It is at this phase of the algorithm where those elements previously ruled out are taken into consideration again.

For instance, in the case of functions, the translation will consist in loading the value of the input variables, executing the function and loading the result obtained into the output variables (considering all diagram elements, i.e., including those which were not considered as nodes in the first phase). Concerning

function blocks, the process is analogous, but it is also necessary to use auxiliary variables to store the information relative to the inputs and outputs of the block. In the case of "LD nodes" (contacts and coils), it is necessary to evaluate the element itself and its input value, since every possible situation has its own translation.

It is, therefore, essential to take every node's type into account, since they would have different translations depending on it. A translation example is shown in TABLE 1 and TABLE 2 . Every LD element needs an associated variable *v*. These tables also show the output auxiliary variables (with '_').

TABLE 1 FBD ELEMENTS WITH THEIR IL TRANSLATIONS.

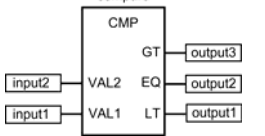
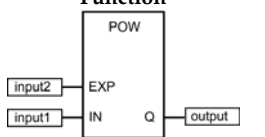
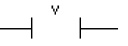
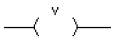
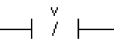
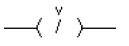
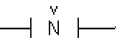
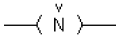


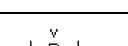
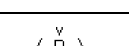
Element	Translation
Function Block compare 	ld input1 st compare.VAL1 ld input2 st compare.VAL2 cal compare ld compare.LT st output1 ld compare.EQ st output2 ld compare.ST st output3
Function POW 	ld input1 POW input2 st POW 1

TABLE 2 LD ELEMENTS WITH THEIR IL TRANSLATIONS

Element	Translation	Element	Translation
	ld _state and v st _state		ld _state st v
	ld _state andn v st _state		ld _state stn v
	ldn v and _prevv and _input st _input ld v st _prevv		ldn _input and _previn st v ld _input st _previn
	ldn _prevv and v and _input st _input ld v st _prevv		ldn _previn and _input st v ld _input st _previn
	ldn _state and v st v		ld _state or v st v

Designed Architecture

As it has previously been stated that the proposed solution in the HEPICA environment, is based on the IEC 61131-3 standard and the TC6 scheme, providing a developmental environment for automation control programs integrated into PLC and softPLC.

HEPICA has been implemented using the .NET platform, which offers flexibility and extensibility through powerful development environments oriented towards its different programming languages. This environment has been implemented in a modular way using C# language. Each intern block is independent, so that its treatment, complexity and configuration can be isolated, reducing in this way its development cost and therefore, making it easier to update. FIG. 9 HEPICA'S MODULAR STRUCTURE.shows the modules (also called blocks) which make up HEPICA's general architecture.

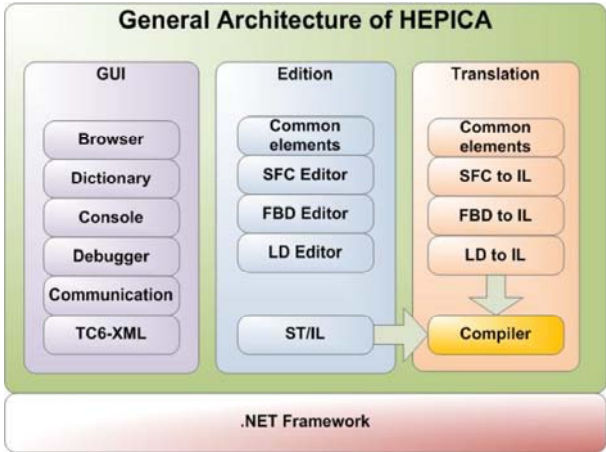


FIG. 9 HEPICA'S MODULAR STRUCTURE.

The Graphical User Interface (GUI) is related to the most general aspects of the environment such as the configuration and storage of projects, or the visualization and debugging of the current state. The edition block includes all those aspects aimed to assist in the design of graphic and textual languages. The translation block focuses on the verification and transformation of graphic programs into IL code. Internally, HEPICA includes a compiler designed to generate executable code compatible with commercial soft PLCs. This compiler has been implemented using the open source tools Flex and GNU Bison.

In the following subsections, each of the independent modules of HEPICA's architecture is described in details.

Graphical User Interface

The GUI of the application incorporates the following components: Solution explorer, debugger, graphical editors and the output console (See fig. 10).

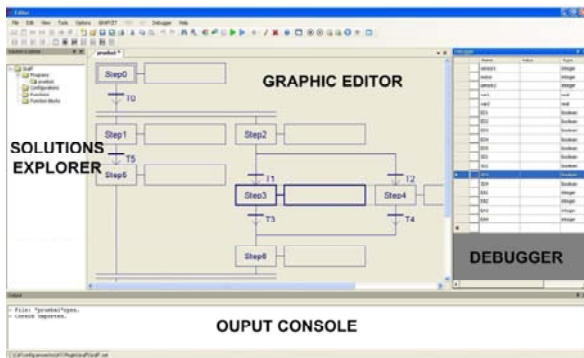


FIG. 10 GRAPHICAL USER INTERFACE OF HEPICA

The solution explorer describes the project's structure. Through this component, the configuration and organisation of the POU associated to each project can be modified.

The debugger covers every execution flow of a program and allows the user to check the current state of all the variables included in it. It also permits programmers to visualize graphically the execution flow of the graphic program that is being debugged, independently of its type.

The edition area establishes a canvas to design graphic diagrams. For textual languages, it sets up a text processor which provides a reserved-word stressing utility.

Through the console, the user is informed about the events that take place. These events can notify the programmers of errors detected during both the verification and the compilation phase, report about the current state of the application or provide any other relevant information about the project being executed.

There exist in many other modules accessible to the toolbar menu, such as the data dictionary or the POU configuration. The first one allows the user to define the variables that make up the project. While the second one establishes the characteristics of a POU according to its type (program, function or function block).

Translators

Translation algorithms generate equivalent IL code from any type of graphic language. Prior to their translation, a verification of the diagrams is done in order to assure that all of them are well designed and

structured. The subsequent translation allows the programmer to represent every single program as a source code file structured according to the specifications of the standard. Mostly, this code will be written in IL language.

In the last few years, the implementation of algorithms to translate graphic languages into IL has become more popular (Huang, Liu, & Liu, 2009; Yan & Zhang, 2008, 2010). This is due to the fact that IL is the most powerful and complete language defined by the standard. This advisable translation is also a required process in HEPICA. In order to be able to process different programming languages, the compiler has been designed to accept only IL and ST as input languages, since it is the best and simplest solution. Therefore, the translator is also in charge of preparing every program for its succeeding compilation. Thus, the translation of a program constitutes a previous phase to its compilation. The algorithms implemented for this purpose have already been described previously in section 4.

Compilation

The HEPICA'S compiler has been developed in C, and makes use of Flex and Bison. It involves the phases of lexical, semantic and syntactic analysis as well as code generation. All of these parts are strongly connected with the error handler and symbols table.

As intermediate language, the compiler uses enriched IL, which is then translated into machine code. This intermediate language adds useful semantic information to the succeeding phases of analysis and code generation.

The features implemented in HEPICA, such as the declaration of functions and function blocks (both standard and user-defined), or the use of local and global variables make it possible to develop very complex programs.

A two-pass parsing is used for the recognition of POU interfaces. During the first one, the scope of every declaration and initialisation is analysed. The intention of the second pass is to examine all the instructions, variables, parameters and other POU invocations in order to check their correct use.

The developed compiler permits programmers to include ST code within an IL program by means of *pragmas*. Pragmas are compiler directives which can be used to intercalate code fragments in different languages.

Finally, during the error detection phase, syntax, type's concordance, scopes and duplicity of the variables, as well as the correct use of every jump instruction to its respective label are checked.

Conclusions

In this paper a new integrated development environment, HEPICA, having been presented for industrial PLCs and softPLCs has been designed according to the standard 61131-3 and following the XML-TC6 scheme, including edition, verification, translation and debugging processes for all the programming languages defined in the standard. Moreover, it is flexible and easily extensible due to its implementation based on the NET platform.

During the translation process, HEPICA implements an original algorithm capable to translate graphic languages into IL, taking advantage of the information supplied by the TC6-XML scheme. Thus, it is not necessary to create new data structures to store relevant information about the diagrams that need to be translated, which is an advantage over all current translation algorithms.

Internally, HEPICA uses a compiler based on Flex and Bison that generates executable code for commercial soft PLCs, easily adaptable to any other soft PLC. To sum up, HEPICA represents a solid initial core with the intention of the achievement of an agile and mature environment in the scope of industrial automation programming.

ACKNOWLEDGMENT

This work has been supported by the Avanza Competitividad R&D Project TSI-020100-2010-484 of the Spanish Ministry of Industry and Commerce, led by Telvent Energia S.A.

REFERENCES

- Association, Plc. (n.d.). PLCopen. *PLCopen TC6 XML specification*. Retrieved from www.plcopen.org
- Chambers, C., Holcombe, M., & Barnard, J. (2001). Introducing X-machine models to verify PLC ladder diagrams. *Computers in Industry*, 45(3), 277–290. doi:10.1016/S0166-3615(01)00085-9
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Section 23.4: Topological sort. *Introduction To Algorithms* (pp. 485–488). MIT Press.
- Esteez, E., Marcos, M., & Irisarri, E. (2009). Analysis of IEC 61131-3 Compliance through PLCope XML interface. *2009 7th IEEE International Conference on Industrial Informatics* (pp. 757–762). IEEE. doi:10.1109/INDIN.2009.5195898
- Fen, G., & Ning, W. (2006). A transformation algorithm of ladder diagram into instruction list based on AOV digraph and binary tree. *TENCON 2006. 2006 IEEE Region 10 ...*, 00, 1–4. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4142169
- Hu, F., Fu, L., Liu, L., & Zhang, G. (2008). An Algorithm about Transforming PLC Ladder Diagram to Instruction List Based on Series-Parallel Merging Method. *2008 IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, 812–816. Ieee. doi:10.1109/PACIIA.2008.13
- Huang, L., Liu, W., & Liu, Z. (2009). Algorithm of transformation from PLC ladder diagram to structured text. *2009 9th International Conference on Electronic Measurement & Instruments*, 4–778–4–782. Ieee. doi:10.1109/ICEMI.2009.5274701
- John, K. H., & Tiegelkamp, M. (2010). *IEC 61131-03: Programming Industrial Automation Systems. Media* (p. 390). Springer Berlin Heidelberg. doi:10.1007/978-3-642-12015-2
- KW Software. (2012). Multiprog. *KW Software*. Retrieved from <http://www.kw-software.com/com/index1024.html>
- Lee, G., Zandong, H., & Lee, J. (2004). Automatic generation of ladder diagram with control Petri Net. *Journal of Intelligent Manufacturing*, 245–252. Retrieved from <http://www.springerlink.com/index/R7M2470764520X3P.pdf>
- Molina, F., Barbancho, J., Leon, C., Molina, A., & Gomez, A. (2007). Using industrial standards on PLC programming learning. *Control Automation, 2007. MED '07. Mediterranean Conference on* (pp. 1–6). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4433652
- Öhman, M., Johansson, S., & Årzén, K.-E. (1998). Implementation aspects of the PLC standard IEC 1131-3. *Control Engineering Practice*, 6(4), 547–555. doi:10.1016/S0967-0661(98)00054-9
- Plc, A., & Engineering, C. (1992). Translating unrestricted

- relay ladder logic into Boolean form, 20, 45–61.
- Thapa, D., Park, C. M., Park, S. C., & Wang, G.-N. (2009). Auto-generation of IEC standard PLC code using t-MPSG. *International Journal of Control, Automation and Systems*, 7, 165–174.
- Tisserant, E., Bessard, L., & De Sousa, M. (2007). An Open Source IEC 61131-3 Integrated Development Environment. 2007 5th IEEE International Conference on Industrial Informatics, 183–187. Ieee. doi:10.1109/INDIN.2007.4384753
- Welch, J.T. (1997). Translating relay ladder logic for CCM solving. *IEEE Transactions on Robotics and Automation*, 13(1), 148–153. doi:10.1109/70.554356
- Welch, John T. (1995). An event chaining relay ladder logic solver. *Computers in Industry*, 27(1), 65–74. doi:10.1016/0166-3615(95)00008-R
- Yan, Y., & Zhang, H. (2008). A New Translation Algorithm from Ladder Diagrams to Instruction Lists. *IFAC Proceedings Volumes (IFAC-PapersOnline)* (pp. 14804–14809).
- Yan, Y., & Zhang, H. (2010). Compiling Ladder Diagram into Instruction List to comply with IEC 61131-3. *Computers in Industry*, 61(5), 448–462. Elsevier B.V. Retrieved from <http://linkinghub.elsevier.com/retrieve/pii/S0166361509002188>